10/719,443 AFV-892

(12) UK Patent Application (19) GB (11) 2 405 506 (13) A

(43) Date of A Publication 02.03.2005

(21) Application No: 0418969.2

(22) Date of Filing: 25.08.2004

(30) Priority Data:
(31) 2003058748 (32) 25.08.2003 (33) KR

(71) Applicant(s):
Samsung Electronics Co., Ltd.,
(Incorporated in the Republic of Korea)
416 Maetan-dong, Yeongtong-gu,
Suwon-si, Gyeonggi-do, Republic of Korea

(72) Inventor(s):
Woo-Hyong Lee

(74) Agent and/or Address for Service:
Marks & Clerk
90 Long Acre, LONDON, WC2E 9RA,
United Kingdom

(51) INT CL[7]:
G06F 11/34 12/02

(52) UK CL (Edition X):
G4A AFMD

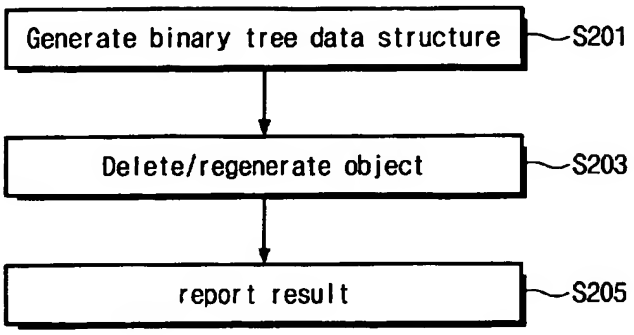
(56) Documents Cited:
WO 2001/073556 A1 WO 2000/041079 A3
US 6118940 A US 6070173 A
Wikipedia, "Binary Search Tree", earliest version 14 June 2002, found at http://en.wikipedia.org/wiki/Binary_search_tree

(58) Field of Search:
UK CL (Edition W) G4A
INT CL[7] G06F
Other: EPODOC, WPI, JAPIO, INSPEC, Internet

(54) Abstract Title: **Benchmarking Garbage Collection in Java Virtual Machines**

(57) In a benchmark file, generating a first plurality of objects in binary tree 201, deleting these objects and replacing with a second plurality of objects 203, executing the benchmark file and reporting at least one result of the execution 205, for example the size of allocated memory, size of unallocated memory, or the time elapsed during the benchmark file execution.

Fig. 2



GB 2 405 506 A

Fig. 1

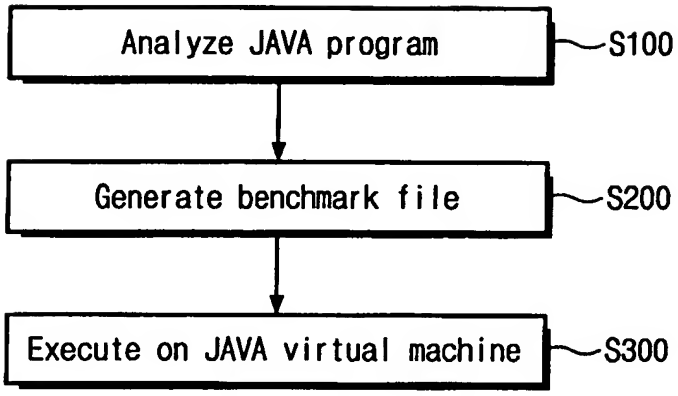

Analyze JAVA program
S100
Generate benchmark file
S200
Execute on JAVA virtual machine
S300

Fig. 2

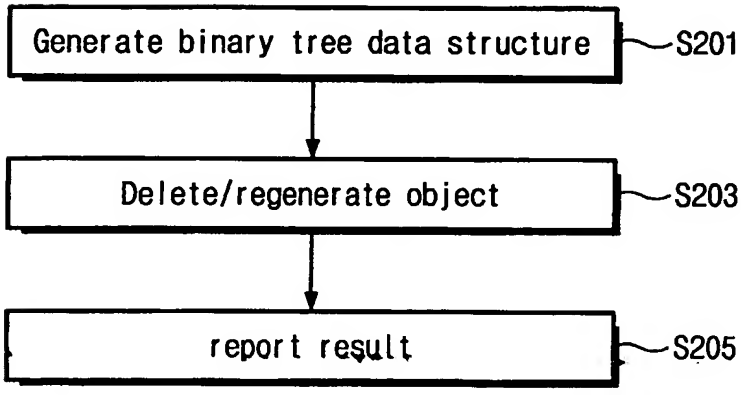

Generate binary tree data structure
S201
Delete/regenerate object
S203
report result
S205

Fig. 3

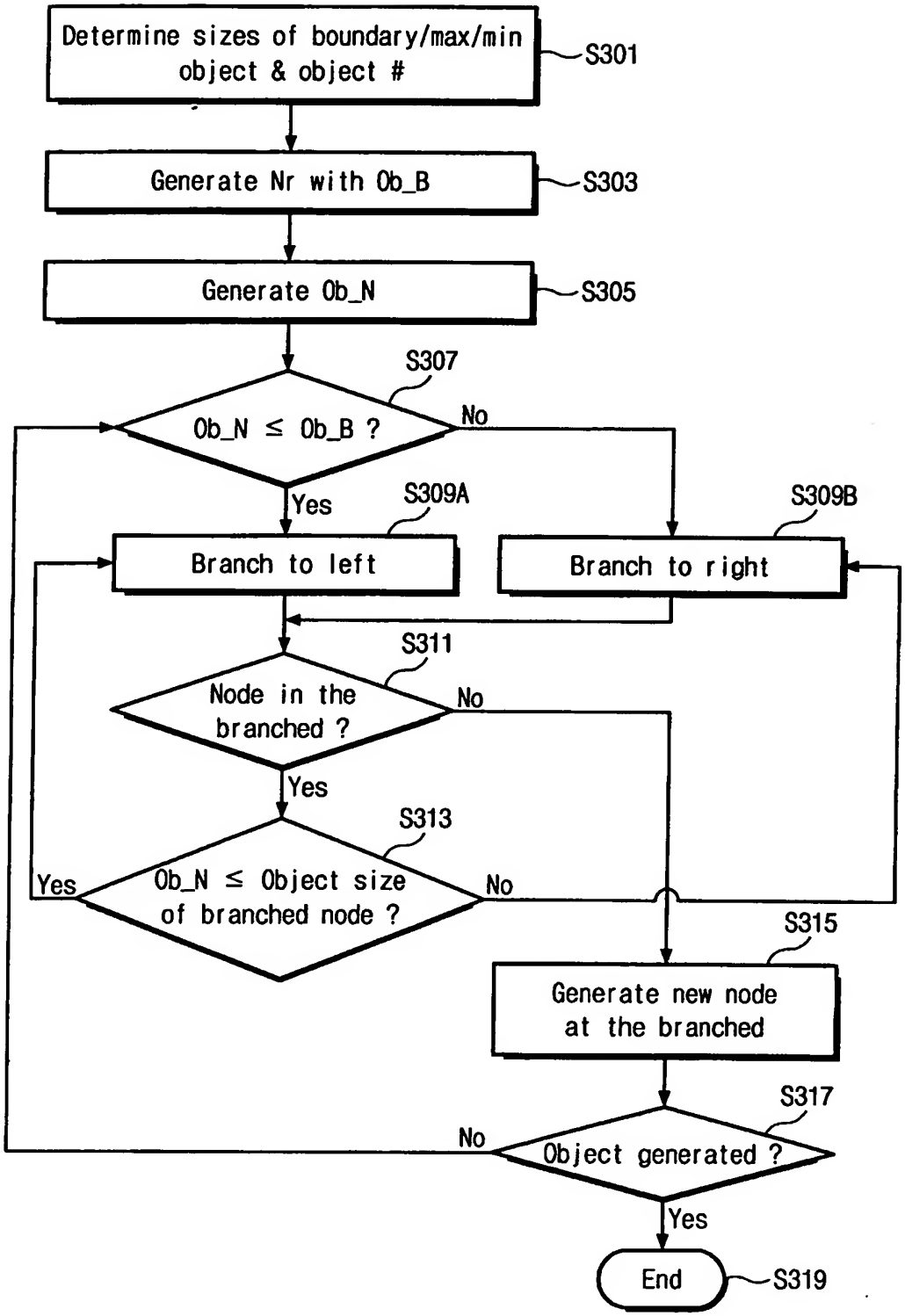

Determine sizes of boundary/max/min object & object #
S301
Generate Nr with Ob_B
S303
Generate Ob_N
S305
S307
Ob_N ≤ Ob_B ?
No
Yes
S309A
Branch to left
S309B
Branch to right
S311
Node in the branched ?
No
Yes
S313
Ob_N ≤ Object size of branched node ?
Yes
No
S315
Generate new node at the branched
S317
Object generated ?
No
Yes
End
S319

Fig. 4A

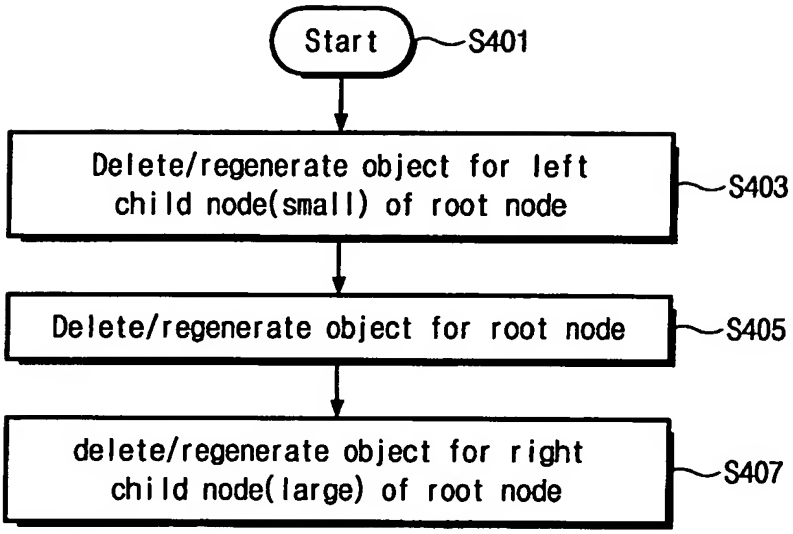

Start
S401
Delete/regenerate object for left child node(small) of root node
S403
Delete/regenerate object for root node
S405
delete/regenerate object for right child node(large) of root node
S407

Fig. 4B

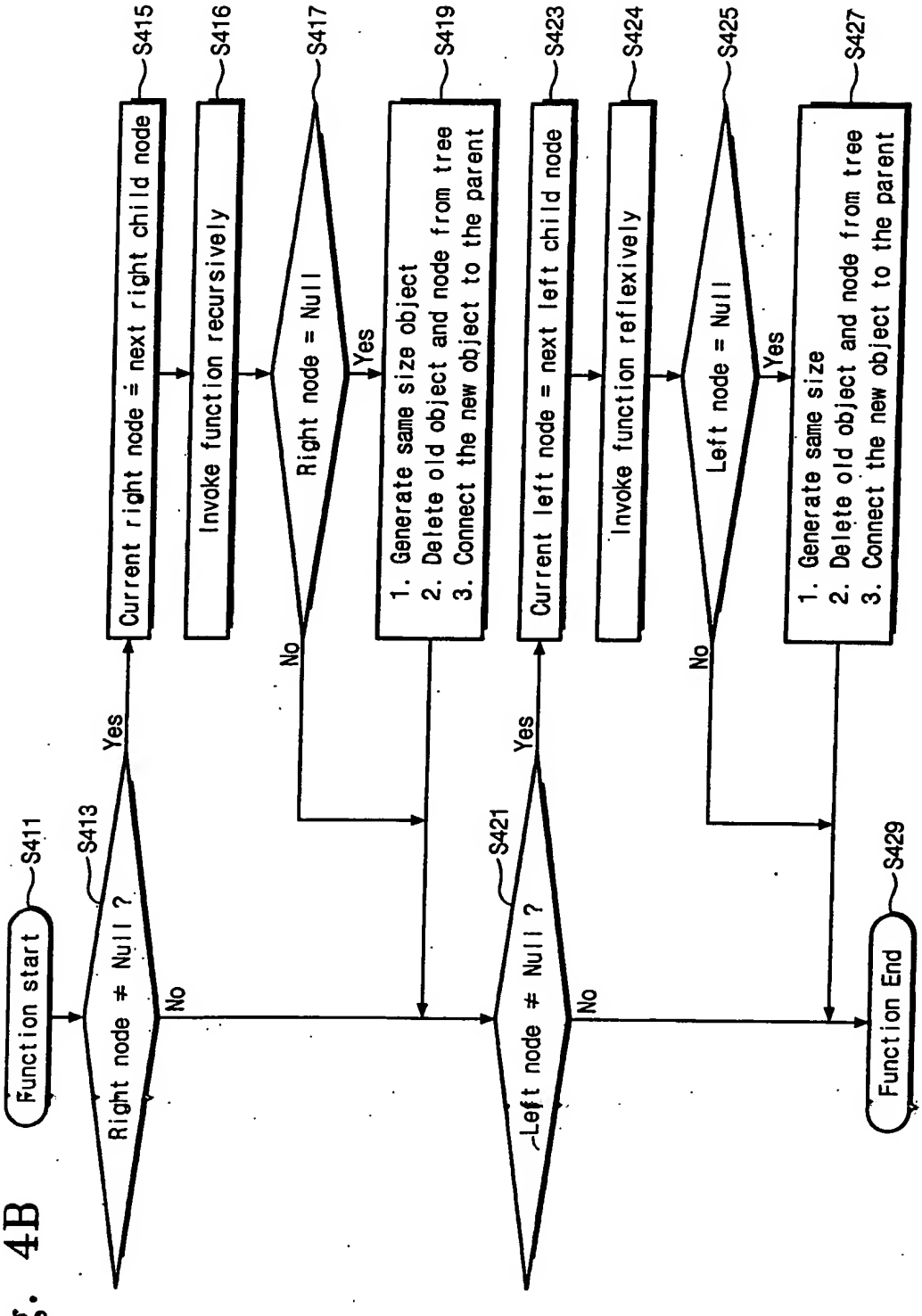

Function start
S411
Right node ≠ Null ?
S413
Yes
No
Current right node = next right child node
S415
Invoke function recursively
S416
Right node = Null
S417
Yes
No
1. Generate same size object
2. Delete old object and node from tree
3. Connect the new object to the parent
S419
Left node ≠ Null ?
S421
Yes
No
Current left node = next left child node
S423
Invoke function reflexively
S424
Left node = Null
S425
Yes
No
1. Generate same size
2. Delete old object and node from tree
3. Connect the new object to the parent
S427
Function End
S429

Fig. 5A

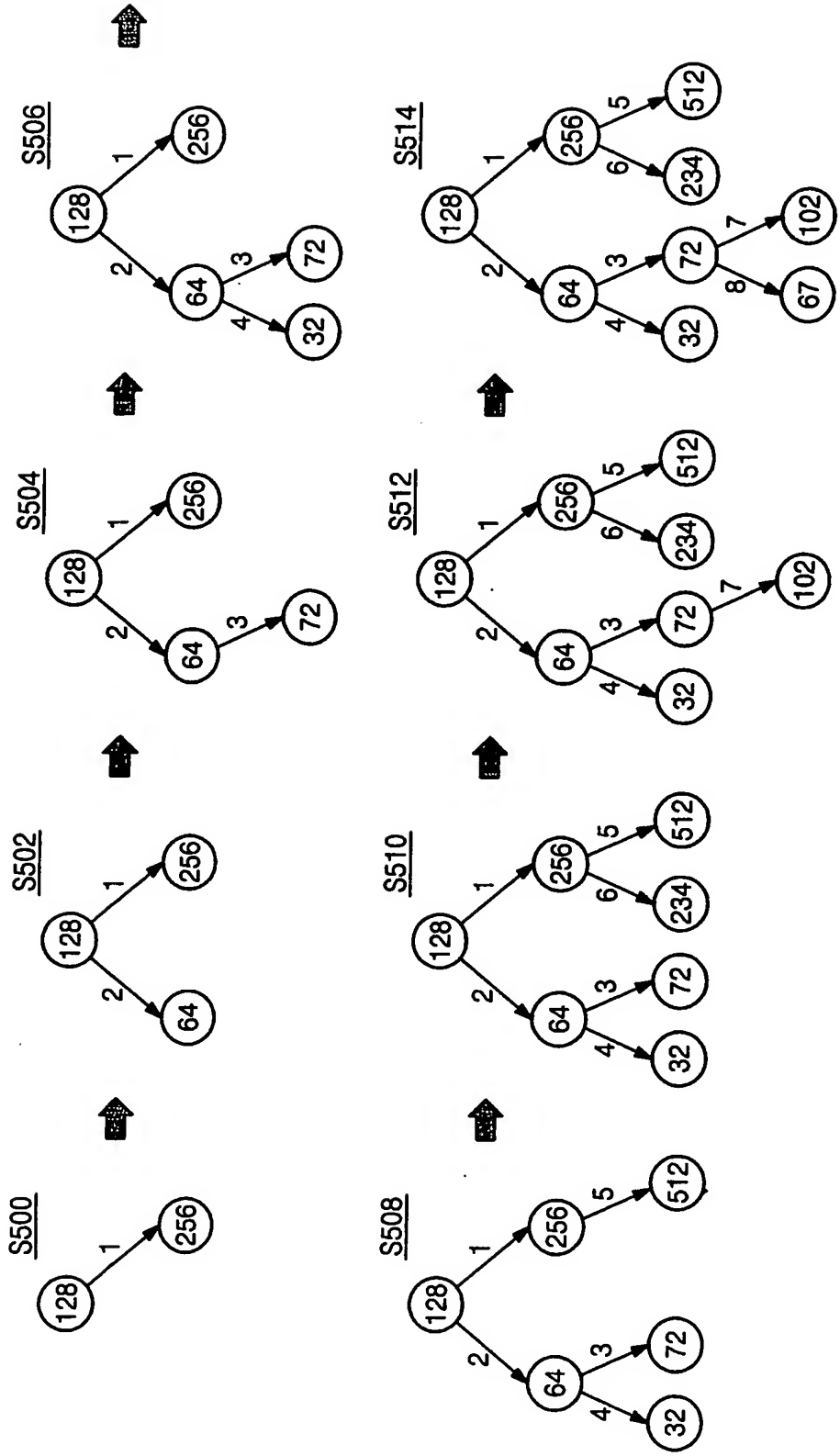

S500
128
1
256
S502
128
1
256
2
64
S504
128
1
256
2
64
3
72
S506
128
1
256
2
64
3
72
4
32
S508
128
1
256
5
512
2
64
3
72
4
32
S510
128
1
256
5
512
2
64
3
72
6
234
4
32
S512
128
1
256
5
512
2
64
3
72
7
102
6
234
4
32
S514
128
1
256
5
512
2
64
3
72
7
102
6
234
4
32
8
67

Fig. 5B

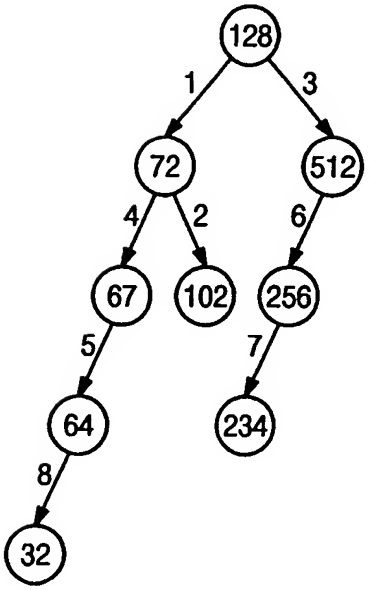


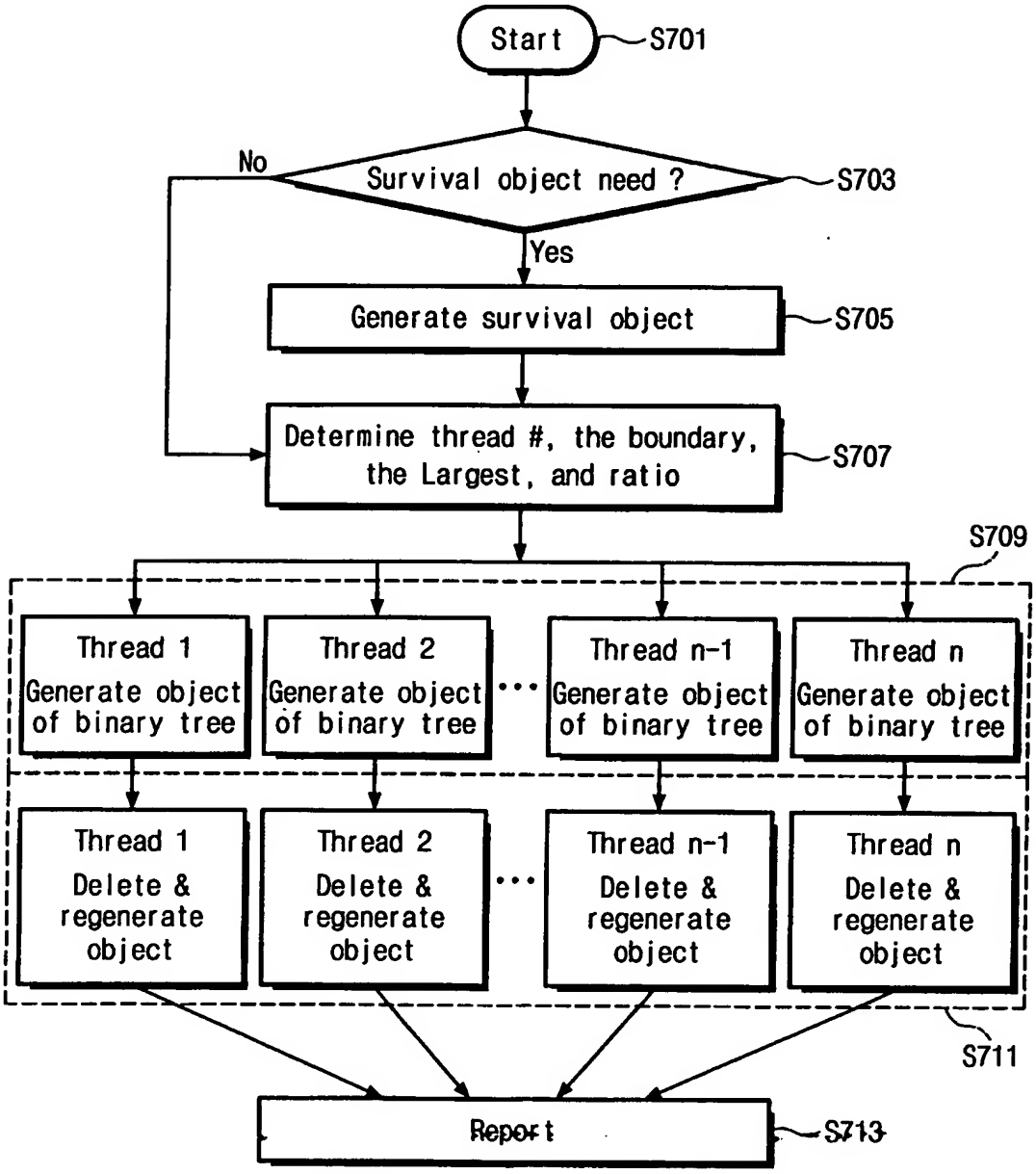
Fig. 6A

Execution Step	The order of deletion & regeneration
1st	102, 67
2nd	102, 67, 72, 32
3rd	102, 67, 72, 32, 64
4th	102, 67, 72, 32, 64, 128
5th	102, 67, 72, 32, 64, 128, 512, 234
6th	102, 67, 72, 32, 64, 128, 512, 234, 256

Fig. 6B

Execution Step	The order of deletion & regeneration
1st	32
2nd	32, 64
3rd	32, 64, 102, 67
4th	32, 64, 102, 67, 72
5th	32, 64, 102, 67, 72, 128
6th	32, 64, 102, 67, 72, 128, 234
7th	32, 64, 102, 67, 72, 128, 234, 256
8th	32, 64, 102, 67, 72, 128, 234, 256, 512

Fig. 7


Start
S701
No
Survival object need ?
S703
Yes
Generate survival object
S705
Determine thread #, the boundary, the Largest, and ratio
S707
S709
Thread 1
Generate object of binary tree
Thread 2
Generate object of binary tree
Thread n-1
Generate object of binary tree
Thread n
Generate object of binary tree
Thread 1
Delete & regenerate object
Thread 2
Delete & regenerate object
Thread n-1
Delete & regenerate object
Thread n
Delete & regenerate object
S711
Report
S713

SYSTEM OF BENCHMARKING AND METHOD THEREOF

[0001] BACKGROUND OF THE INVENTION

Field of the Invention

[0002] The present invention, in general, relates to benchmarking, and more particularly, to a method and system for benchmarking garbage collection in a Java virtual machine.

Discussion of the Related Art

[0003] Java is emerging as a common programming language which may be used on many different types of platforms. Java source code may be compiled to be executed on a Java virtual machine (JVM). A platform equipped with a JVM may execute Java source code. Java source code may be executed by compiling Java source code into Java byte code and then having the JVM execute the compiled Java byte code. Java byte code may be regarded as machine language associated with the JVM. The Java byte codes may be sequences of instructions which may be executed with the JVM.

[0004] The JVM may be a virtual computing system installed on a hardware platform with an operating system. The JVM may compile source code into Java byte code which does not need to be re-compiled for execution on other hardware platforms and/or operating systems. Instead, once compiled, Java byte code may be executed on any platform containing a JVM.

[0005] The JVM may be classified into four sections: a register, a stack, a garbage-collected heap, and a function field. The function field may be a region where Java byte codes may be permanently stationed. The stack may store parameters for Java byte code commands and/or

results of the execution of the Java byte code commands, and may transfer the parameters to the function field. The stack may receive parameter values from the function field, and may retain an invocation state for each function. The garbage-collected heap may be a free storage space containing Java program objects. A memory may be generated from the garbage-collected heap with memory allocation by using an operator "NEW". The references to Java program objects on the garbage-collected heap may be traced in a Java runtime environment.

[0006] In the JVM, Java program objects may be dynamically allocated from the garbage-collected heap and returned to a system. A garbage collection algorithm may determine accessible Java program objects. When it is not possible to access a Java program object, a memory location occupied by the Java program object may return to the system where it may be reused even though it was not de-allocated.

[0007] Java program objects may be continuously generated and garbage collection may be performed during the execution of the Java byte code. The Java garbage collection may be one of the most time consuming operations performed in the JVM. Therefore, the performance of the JVM may be affected by garbage collection. While JVMs on different platforms may represent various levels of garbage collection performance, most JVM benchmarks are based on transactions between a client and a central processing unit and/or a server.

[0008] Therefore, specifically benchmarking the garbage collection of a JVM may be difficult when benchmarking according to conventional methods. For example, a first JVM on a first platform may have better garbage collecting performance than a second JVM on a second platform, but if the second JVM on the second platform had superior performance overall, one skilled in the art may reach the erroneous conclusion that the second JVM on the second platform also had superior garbage collection performance.

SUMMARY OF THE INVENTION

[0009] An exemplary embodiment of the present invention is a method of benchmarking, including generating a first plurality of objects within a benchmark file, generating a second plurality of objects within the benchmark file which do not reference the first plurality of objects, executing the benchmark file, and reporting at least one result of the execution.

[0010] Another exemplary embodiment of the present invention is a system of benchmarking, including a processor for generating a first plurality of objects of a first binary tree data structure in a benchmark file for generating a second plurality of objects of a second binary tree data structure without reference to the first binary tree data structure in the benchmark file, a processor for executing the benchmark file, and for reporting a result of the execution.

[0011] Another exemplary embodiment of the present invention is a computer program including a computer-readable medium having computer program logic stored thereon for enabling a data processing unit to perform benchmarking, including generating a first plurality of objects of a first binary tree data structure in a benchmark file, generating a second plurality of objects of a second binary tree data structure in the benchmark file, executing the benchmark file, and reporting at least one result of the execution.

BRIEF DESCRIPTION OF THE DRAWINGS

[0012] The present invention will become more apparent by describing, in detail, exemplary embodiments thereof with reference to the attached drawings, wherein like elements are represented by like reference numerals, which are given by way of illustration only and thus do not limit the exemplary embodiments of the present invention.

[0013] FIG. 1 illustrates a benchmarking process of Java garbage collection according to an exemplary embodiment of the present invention.

[0014] FIG. 2 illustrates a process of generating a benchmark file.

[0015] FIG. 3 illustrates a process of generating the objects of a binary tree data structure according to an exemplary embodiment of the present invention.

[0016] FIGS. 4A and 4B illustrate the deletion and regeneration as applied to objects in a binary tree data structure according to an exemplary embodiment of the present invention.

[0017] FIGS. 5A and 5B illustrate a mechanism of forming objects nodes of the binary tree data structure according to an exemplary embodiment the present invention.

[0018] FIGS. 6A and 6B illustrate the sequence of deleting and regenerating objects in the binary tree data structures shown in FIGS. 5A and 5B, respectively, according to exemplary embodiments of the present invention.

[0019] FIG. 7 illustrates a process of generating benchmark files according to another exemplary embodiment of the present invention.

DETAILED DESCRIPTION OF EXEMPLARY EMBODIMENTS OF THE INVENTION

[0020] Hereinafter, exemplary embodiments of the present invention will be described in detail with reference to the accompanying drawings.

[0021] In an exemplary embodiment of the present invention, the ~~procedure of~~ workload and benchmarking for garbage collection is exemplarily operable in Java virtual machines (JVMs), but the present invention may be applicable in other programming languages and/or other virtual machines (VMs).

[0022] In an exemplary embodiment of the present invention, the execution JVM may include a binary tree data structure. Binary tree

data structures are well known in the art and will not be discussed hereafter.

[0023] In an exemplary embodiment of the present invention, a maximum depth of a binary tree data structure may be interpreted as a depth of a child node most apart from a root node. For example, assuming that a depth of a root node is 1, two child nodes of the root node may be established to have the deepest depth of 2 wherein no node exists with a depth greater than 2.

[0024] FIG. 1 illustrates a benchmarking process of Java garbage collection according to an exemplary embodiment of the present invention. In S100, a Java program may be analyzed, specifically with regard to an object size, an object lifespan, a size of a largest object, a size of a smallest object, a number of objects, a size of an object frequently generated, and/or other object characteristics.

[0025] In S200, a benchmark file may be generated. The benchmark file may contain information about a generation, deletion, and regeneration of an object based on the analysis in S100. Smaller objects may have a shorter lifespan than larger objects because the smaller objects may be more frequently generated and deleted. The process of generating, deleting, and regenerating objects will be described below.

[0026] In S300, evaluating a garbage collection of JVM may be accomplished by executing benchmark files on various JVMs, in which a benchmark file which contain a *.java file and/or a *class file. The benchmark file may be converted into Java byte code and thereafter the converted file may be executed on the JVM.

[0027] FIG. 2 illustrates a process of generating a benchmark file. In S201, objects may be generated in a binary tree data structure. During object generation in the binary tree data structure, a root node may be generated with a size an average size based on the analysis of S100. The root node may be referred to as a boundary object. The root node may also be representative of an intermediate size between the

largest and smallest objects based on the analysis of S100. A child node diverging from the root node, and/or a parent node, may have an intermediate size between the largest and smallest objects based on the analysis of S100. Objects of a size less than or equal to a size of an object of the root node are branched left, while objects larger than the boundary object are branched right. It should be understood that a first node is larger than a second node when a first object in the first node is larger than a second object in the second note. Branching with respect to binary tree data structures is well known in the art and will not be discussed further. In addition, when child nodes diverging from a parent node are less than or equal to their parent node, the child nodes may be branched left from their parent node, while child nodes larger than their parent node are branched right from their parent node. A ratio of the numbers of objects branched left and right from the root node may be adjusted. For example, the size of the root node may be adjusted in order to change the ratio.

[0028] FIG. 3 illustrates a process of generating the objects of a binary tree data structure according to an exemplary embodiment of the present invention.

[0029] In S301, the sizes of the boundary object, the largest object, smallest object, and/or the number of objects (i.e., the number of nodes) may be determined.

[0030] In S303, the root node may be generated with the boundary object Ob_B. The boundary object Ob_B may be generated with a determined size, forming the root node Nr.

[0031] In S305, a new object Ob-N may be generated, wherein Ob-N ~~may have~~ a random object size in a range from the size of the smallest object to the size of the largest object. While adding new objects to the binary tree data structure, a ratio of the numbers of the new objects less than the boundary object to the number of new objects larger than the boundary object may be determined.

[0032] In S307, the new objects generated may be compared to the boundary object. From the comparison with the new objects, if one of the new objects, Ob-N, is less than or equal to the boundary object Ob_B, then Ob-N may be branched left from the root node and/or the parent node in S309A. If Ob-N is larger than the boundary object Ob_B, then Ob-N may be branched right from the root node and/or the parent node in S309B.

[0033] In S311, a node in the branched side may be analyzed. If there is already an existing node in the branched side, S313 compares the existing node in the branched side with Ob-N. After the comparison with the object of the existing node in the branched side and Ob-N, if the new object Ob_N is less than or equal to the object of the existing node in the branched side, the process may advance to S309A. Otherwise, if Ob-N is larger than the object of the existing node in the branched side, the process may advance to S309B. S311 may only advance to S315 when there is no node in the branched side.

[0034] In an exemplary embodiment of the present invention, the process may advance to S315 to generate a new node for the branched side when there is no existing node in the branched side.

[0035] In S317, the process may return to S307 when a new object has not been generated in the branched side. The process may advance to S319 from S317 after generating the new object Ob_N in the binary tree data structure.

[0036] Referring to FIG. 2, after the binary tree data structure is generated in S201, the new node generated in S317 of Fig. 3 may be deleted and renewal objects may be generated in S203. Deletion and ~~regeneration~~ may first be performed on nodes which branched left from the root node. Deletion and regeneration may then be performed on the root node. Deletion and regeneration may then be performed on nodes which branch right from the deleted root node.

[0037] The deletion and regeneration of objects in each branch side may progress from the deepest depth to the shallowest depth. For

example, it may be assumed that the binary tree data structure is formed with a depth of "4". In this example, with respect to the left branch of the root node, the deletion and regeneration may progress in the order of sibling nodes with depth 4, 3, 2, and 1 in sequence. Following the deletion and regeneration of the left branch, the root node with depth 1 may be deleted and regenerated. Following the deletion and regeneration of the root node, the deletion and regeneration of the right branch may progress in the order of sibling nodes with depth 4, 3, 2, and 1 in sequence.

[0038] In an exemplary embodiment of the present invention, the order of the deletion and regeneration for nodes may be represented by 4_L, 3_L, 2_L, 1, 4_R, 3_R, and 2_R, wherein the numbers represent the depth of the nodes, and the bottom characters L and R represent the branched sides with respect to the root node, where L represents the left branch and R represents the right branch. Each node may comprise up to two sibling nodes. The deletion and regeneration may be progress from the right branch side to the left branch side as discussed above.

[0039] The regeneration may be performed with an object that has the same size as a corresponding deleted object. By regenerating an object of a same size, an identical binary tree data structure may be maintained.

[0040] In an exemplary embodiment of the present invention, the deletion and regeneration for an object in a node of a first depth may be executed after completing the deletion and regeneration for another object in a node of a second depth wherein the second depth may be higher than the first depth. In this exemplary embodiment, the deletion and regeneration applied to objects with nodes may be further applied to all of the objects child nodes.

[0041] In an exemplary embodiment of the present invention, the deletion and regeneration for nodes of right branch from the root node may be performed after the deletion and regeneration may be applied to

the nodes of the left branch from the root node. For example, with the binary tree data structure with a depth 4, wherein the root node depth is 1, the deletion and regeneration applied to objects may be executed in the order of (4_L), (4_L, 3_L), (4_L, 3_L, 2_L), (4_L, 3_L, 2_L, 1), (4_L, 3_L, 2_L, 1, 4_R), (4_L, 3_L, 2_L, 1, 4_R, 3_R), (4_L, 3_L, 2_L, 1, 4_R, 3_R, 2_R) where the deletion and regeneration is applied first to 4_L and last to 2_R. Therefore, the smaller objects may be more frequently deleted and regenerated, while the larger objects may be less frequently deleted and regenerated. In order to retain the binary tree data structure prior to the deletion and regeneration, the regeneration may generate new child nodes having the same sizes as the deleted child nodes.

[0042] FIGS. 4A and 4B illustrate the deletion and regeneration as applied to objects in a binary tree data structure according to an exemplary embodiment of the present invention.

[0043] Referring to FIG. 4A, the deletion and regeneration may be initially applied to nodes on the left branch from the root node in S403. The deletion and regeneration may be applied to the root node in S405. The deletion and regeneration may be applied to nodes on the right branch from the root node in S407.

[0044] In order to delete a node and regenerate a new node for the deleted node, binary tree searching may be carried out in advance. In other words, after searching a highest depth node for a present node, deletion and regeneration may be executed in the order from the highest (i.e., deepest) depth node to the present node. A parent node of the present node may be set as a next present node. This process may repeat from the left nodes to the root node, the root node, and then to the right nodes to the root node.

[0045] Referring to FIG. 4B, the regeneration begins in S411. In S413, it is determine whether the child node of the right branch has a null state. A node in a null state does not contain an object. If the right child node has a null state, the process may advance to S421 to determine whether the child node of the left branch is in a null state. If

the left child node has a null state, the execution of the regeneration may be terminated in S429.

[0046] In S413, if a right child node does not have a null state, the process may advance to S415. In S415, the current right child node may be set as the next right child node and the regeneration of FIG. 4B may begin for the next right child node. The process of FIG. 4B may be executed recursively and may repeat until it has been executed for all nodes in the binary tree data structure in S416. After executing the recursive iterations of S416, the process may advance to S417. In S417, it may be determined whether the current right child node has a null state. If the current right child node has a null state, a new object may be generated in the right child node. If the current right child node does not have a null state, the process may advance to S421.

[0047] In S421, if a current left child node does not have a null state, the process begins for the next left child node. The process of FIG. 4B may be executed recursively and may repeat until it has been executed for all nodes in the binary tree data structure in S424. After executing the recursive iterations of S424, the process may advance to S423. In S423, the current left child node may be set as the next left child node and then the process of FIG. 4B may advance to S425. In S425, it is determined whether the current left child node has a null state. If the current left child node has a null state, a new object may be generated in the left child node. If not, the process may be terminated in S429.

[0048] In an exemplary embodiment of the present invention, the process illustrated in FIG. 4B may be adjusted in its order such that the left child node (or the left sibling node) is deleted and regenerated prior to the right child node (or the right sibling node). This may be accomplished by switching S413 and its associated actions with S421 and its associated actions.

[0049] In an exemplary embodiment of the present invention, the binary tree data structure and processes thereof as described above

may be executed by a programmer with an ordinary knowledge of computer programming languages.

[0050] In an exemplary embodiment of the present invention, a benchmarking program may be associated with an ObjBTree class, an MTBinTree class, and a RanGen class. The ObjBTree class may allow for the generation, deletion, and regeneration of nodes. The MTBinTree class may allow for the generation of a binary tree data structure with the functions provided by the ObjBTree class. The RanGen class may allow the functions for counting random numbers for use with the generation of new objects.

[0051] In an exemplary embodiment of the present invention, the ObjBTree class may include an add function, a traverse function, a getDepth function, a getdepth function, and a regenNode function. These functions will be described in detail below.

[0052] A function ObjBTree.add(ObjBTree r, Integer n, int objsize) may generate a new node with a new size to be inserted into a binary tree data structure when there is a request for a generation of the new node, wherein a new node larger than the parent node is set to the right sibling node while a node equal to or smaller than the parent node may be set to the left sibling node. A function ObjBTree.traverse(ObjBTree r) may visit all nodes more than one time. A function ObJBTree.getDepth (ObjBTree) may return the depth of a binary tree data structure and may be called by a function ObjBTree.getdepth(ObjBTree r). A function ObjBTree.regenNode (ObjBTree r, int depth) may visit respective nodes to delete their previous objects (i.e., render the previous objects garbage) and may generate new objects within the nodes with the same size to maintain the same tree structure.

[0053] An example of the above described exemplary embodiment of the present invention is given below. In this example, source code of the function add () may be arranged as follows.

[0054] 1. public static ObjBTree add(ObjB"free r, Integer n, int objsize){

```
2.          if (r = = null) {
3.          r = new ObjBTree();
4.          r.left = rright = null;
5.          r.idata = n;
6.          r.sdata = new StringBuffer();
7.          r.sdata. setLength(obj size);
8.      } else if (r.idacompareTo(n) < 0)
9.          r.right = add(r.right, n, objsize);
10.     else
11.         r.left = add(r.left, n, objsize);
12.         return r;
13.         }
```

[0055] The line numbers 1-13 are listed for purpose of reference only, and do not constitute source code.

[0056] In the above example source code of the add() function, if a current node has a null state as determined in line 2, then the left and right child nodes form a new node with a size (objsize) of the null state. If a current node is not in a null state, an object of the current node may be compared with a new object in line 8. If the new object is larger than an object of the current node, the new object may be formed in a child node by being branched right from the current node (i.e., the parent node). If the new object is less than or equal to an object of the current node, then the new object may be formed in a child node by being branched left from the current node.

[0057] An example of the above described exemplary embodiment of the present invention is given below. In this example, source code of the function getDepth () may be arranged as follows.

```
1.  public void getDepth(ObJBTree r) {
2.          if(r = null) {
3.                  curr_depth++;
4.                  if (curr_depth > tree_depth)
```

```
5.                    tree_depth = curr_depth;
6.                getDepth(r.left);
7.                getDepth(r.right);
8.                curr_depth--;
9.    }
10.   }
```

[0058] The line numbers 1-10 are listed for purpose of reference only, and do not constitute source code.

[0059] A depth of a binary tree data structure may be obtained executing functions getDepth(r.left) and getDepth(r.right) recursively, as shown in lines 6 and 7.

[0060] The function regenNode() may instruct nodes to delete their previous objects (i.e., render the previous objects garbage) and may generate new objects within the nodes with the same size to maintain the same tree structure. An exemplary source code of the regenNode() function may be as follows.

```
1.    public ObjBTree regenNode(ObjBTree r, int depth) {
2.     ObjBTree tmpleft = null, tmpright = null;
3.     int tmpidata = 0;
4.     curr_depth++;
5.     if(r.right ! =null) {
6.          tmpidata = r.right.idata.intValue();
7.          tmpleft = rright.left;
8.          tmpright = r.right.right;
9.          r.right = regenNode(r.right, depth);
10.    if(r.right = = null) {
11.         r.right = new ObjBTree( );
12.         Integer tmpIdata = new Integer(tmpidata);
13.         r.right.idata = tmpIdata;
```

```
14.          r.right.sdata = new StringBuffer( );
15.          r.reft.sdata.setLenghth(tmpright);
16.              r.right.right = tmpright;
17.              r.right.left = tmpleft;
18.     }
19.  }
20.     if(f.left ! = null) {
21.     tmpidata = r.left.idata.intValue( );
22.     tmpleft = r.left.left;
23.     tmpright = r.left.right;
24.          r.left = regenNode(r.left, depth);
25.     if(r.left == null) {
26.          r.left = new ObjBTree( );
27.          Integer tmpIdata = new Integer(tmpidata);
28.          r.left.idata = tmpIdata;
29.          r.left.sdata = new StringBuffer( );
30.          r.left.sdata.setLength(tmpidata);
31.          r.left.right = tmpright;
32.          r.left.left = tmpleft;
33.     }
34.  }
```

[0061] The line numbers 1-34 are listed for purpose of reference only, and do not constitute source code.

[0062] The function regenNode() may contain conditional statements to determine whether the left and right branched nodes are in null states as shown by the lines 5 and 20, respectively. Each ~~conditional~~ ~~statement~~ may also include a statement for executing the regenNode function recursively as shown on the lines 9 and 24, and a nested conditional statement to determine whether or not the left or the right node is in a null state as shown on the lines 10 and 25. If either the left or the right branched node has a null state, a new object

may be generated. All of the objects with the deeper depth under a node of a specific depth may be deleted and regenerated.

[0063] The MTBinTree class may be associated with an add() function, a regenNode() function, a getdepth() function, and/or a run() function. The MTBinTree.add() function may generate nodes until a desired number nodes have been generated. The MTBinTree.getdepth( ) function may determine the deepest depth in a generated binary tree data structure. The MTBinTree.regen( ) function may instruct each node of the generated binary tree data structure to delete and regenerate objects therein.

[0064] The MTBinTree class may also inherit a thread class to generate a multiplicity of binary tree data structures simultaneously. The MTBinTree.run( ) function may include a control statement for determining the number of objects, the largest object, the smallest object, the boundary object (the root node), and/or the ratio of objects larger than the boundary object to objects smaller than the boundary object, and may execute the functions MTBinTree.add(), MTBinTree.getdepth(), and MTBinTree.regen() within the control statement.

[0065] An exemplary embodiment of the source code of the thread class may be as follows.

```
1. class MTBinTree extends Thread {
2.     protected ObjBTree bnode = null;
3.     protected ObjBTree rnode = null;
4.     protected ObjBTree rnode_1 = null;
5.     protected ObjBTree rnode_1 = null;
6.     protected Integer objj = null
7.     protected int currdepth = 0;
8.     protected static int bmaxsize = 0;
9.     protected static int bnodes = 0;
10.    protected static int bweight = 0;
11.    protected static int bsizebound = 0
```

```
12.  MTBinTree( ) { }
13.  public void add( ) {
14.          bnode = bnode.add (bnode, obji, obji.intValue( ) );
15.  }
16.  public void regen (ObjBTree node) {
17.          bnode = bnode.regenNode(node, currdepth);
18.  }
19.  public int getdepth(ObJbTree node) {
20.          pub return bnode.getdepth(node);
21.  }
22.  public void run( ) {
23.          int stringsize = 0;
24.          int removedobjs = 0;
25.          int objsize[ ];
26.          int largeobj = 1
27.          int smallobj = 1;
28.          int treedepth_l = 0, treedepth_r = 0;
29.          RanGen r;
30.          JGCW jgcw_bin = new JGCW ( );
31.          bmaxsize = jgcw_bin,bmaxsize;
32.          bnodes = jgcw_bin.bnodes;
33.          bweight = jgcw_bin,bweight;
34.          bsizebound = jgcw_bin,bsizebound;
35.          objsize = new int[bnodes];
36.          r = new RanGen(bmaxsize);
37.          for (int i = 0; i<bnodes; i++) {
38.          if ((largeobj*bweight) > (smallobj*(10-bweight))) {
39.                       stringsize > bsizebound) {
40.                       while (stringsize > bsizebound) {
41.                       stringsize = r.RInt( );
42.                       }
43.                       objsize[i] = stringsize;
```

```
44.                     smallobj++;
45.             }
46.             else {
47.                     stringsize = r.RInt( );
48.                     while (stringsize > bsizebound) {
49.                     stringsize = r.RInt( );
50.                     }
51.                     objsize[i] = stringsize;
52.                     largeobj++
53.             }
54.     }
55.     Integer objbound = new Integer(bsizwbound);
56.     obji = objbound;
57.     add( );
58.     rnode = bnode;
59.     for (int i = 1; i<bonodes; I++) }
60.             Integer objint = new Integer(objsize[i]);
61.             obji = objint;
62.             add( );
63.     }
64.     jgcw_bin.MemWaterMark( );
65.     rnode_l = rnode.left;
66.     rnode_r = rnode.right;
67.     treedepth_l = getdepth(rnode_l);
68.     treedepth_r = getdepth(rnode_r);
69.             for (int i = treedepth_l; i > 1; i--) {
70.                     currdepth = I;
71.                     regen(rnode_l)
72.             jgcw_bin.MemWaterMark( );
73.     }
74.     for(int i = treedepth_r; i > 1; i--) {
75.             currdepth = 2;
```

```
76.                 regen(mode_1);
77.                 currdepth = i;
78.                 regen(rnode_r);
79.                 jbcw_bin.MemWaterMark( );
80.             }
81.     }
82. }
```

[0066] The line numbers 1-82 are listed for purpose of reference only, and do not constitute source code.

[0067] On lines 13-15, the add() function is defined. The add() function may access 'add(ObjBTree r, Integer n, int objsize)' of the ObjBTree class. On the lines 16-18, the regen() function is defined. The regen() function may access 'regenNode(ObjBTree r, int depth)' of the ObjBTree class. On the lines 19-21, the getdepth() function is defined. The getdepth() function may access 'getdepth(ObjBTree r, int depth)' of the ObJBTree class.

[0068] The run() function may be called on line 22. In the run() function, random number may be generated to form a new node with an object on the lines 36-54. The random number may be set to a value between the smallest object size and the largest object size. As the run() function is executed, objects with sizes both smaller and larger than the boundary object may be generated with some ratio between the smaller and larger objects.

[0069] A size of the boundary object is defined on line 34. The root node may be generated on the lines 55 through 58.

[0070] On lines 59-63, additional nodes (e.g., child nodes to the root node) may be generated by using the random number determined by the run() function.

[0071] On lines 65-68, the child nodes left and right of the root node may be generated. On lines 69-73, the deletion and regeneration may be executed for the left child nodes and the root node, with the 'for'

loop statement performing an iterative function for multiple deletions and regenerations.

[0072] On lines 74-78, the deletion and regeneration for object may be applied to the child node right of the root node. The function 'regen(rnode_l)' may be executed to delete and regenerate the objects of the left child nodes on the line 76, and then the function 'regen(rnode_r) may be executed to delete and regenerate the objects of the right child nodes. Thus, the objects of the child nodes right of the root node may be deleted and regenerated after each of the objects of the child nodes left of the root node may be deleted and regenerated.

[0073] Referring again to FIG. 2, in S205, the result of benchmarking may be reported after at least one operation of the deletion and regeneration of objects in the binary tree data structure. The result of benchmarking may include execution times for allocating objects and re-requesting free objects, memory conditions (heap size, allocated spaces, free space, and etc.), and other benchmarking criteria.

[0074] In an exemplary embodiment of the present invention, the execution times may be obtained by time stamping both the beginning and finishing times of the program. Time stamping is well known in the art and will not be described further. The difference between the beginning and finishing times of the program may represent the execution time of the program.

[0075] FIGS. 5A and 5B illustrate a mechanism of forming objects nodes of the binary tree data structure according to exemplary embodiments of the present invention. In an example, it may be assumed that the sizes of the boundary object, the largest object, and the least object are 128 bytes, 512 bytes, and 1 bytes, respectively. In an example, the number of the objects may be 9, including the boundary object. With the exception of the boundary object, additional objects may be generated at random.

[0076] Referring FIG. 5A, the objects generated in the order of 256, 64, 72, 30, 512, 234, 102, and 67. Referring to FIG. 5B, the objects

may be generated in the order of 71, 102, 512, 67, 74, 256, 234, and 32.

[0077] In an exemplary embodiment of the present invention, referring to FIG. 5A, the root node may be formed for the boundary object with a size of 128 bytes. The first object generated with 256 bytes is larger than the boundary object with a size of 128 bytes, and it is branched right from the root node (i.e., the boundary object) and forms a first right child node with a depth of 2, and a path 1 as illustrated in S500. The second object generated with 64 bytes is smaller than the boundary object of the root node, and it is branched left from the root node and forms a left child node with a depth of 2, and a path 2 as illustrated in S502. The third object is generated with 72 bytes and is smaller than the boundary object of the root node, and it is branched left from the root node and then further branched right from the left child node of 64 bytes (path 3), as it is smaller than the boundary object (128 bytes) of the root node but larger than the second object (64 bytes), forming a right child node with a depth of 3.

[0078] Similar to the process described above with respect to the first, second and third objects, additional objects are added in S506, S508, S510, S512, and S514. The resultant binary tree data structure of S514 contains each inserted object.

[0079] In another exemplary embodiment of the present invention, referring to FIG. 5B, the same objects have been inserted into the illustrated binary tree data structure as FIG. 5A, only in a different order, thus creating a different binary tree data structure. With the generation order different from that of FIG. 5A, the first object generated with 72 bytes is branched left from the root node and then forms a left child node with a depth of 2 (path 1). Similar to the process described above with respect to the first object, additional objects are added to the binary tree data structure of FIG. 5B. The resultant binary tree data structure of FIG. 5B may contain each object inserted into the resultant binary tree data structure.

[0080] FIGS. 6A and 6B illustrate the sequence of deleting and regenerating objects in the binary tree data structures shown in FIGS. 5A and 5B, respectively, according to exemplary embodiments of the present invention. As described above, the deletion and regeneration of objects proceeds in the order of the left branched nodes from the root node, the root node, and the right branched nodes from the root node, and further from the highest depth to the lowest depth. As shown, an object is deleted and then regenerated.

[0081] Referring to FIG. 6A, which corresponds to the binary tree data structure of S514 in FIG. 5A, the objects of 102 bytes and 67 bytes, having the highest depth 4 in the binary tree data structure, among the left child nodes (herein, 102, 67, 72, and 32 bytes) of the root node (128 Byte), may be deleted and regenerated in order. The right sibling nodes (e.g., 102 Byte) may be deleted and regenerated earlier than the left sibling nodes (e.g., 67 Byte), or vice versa. As shown in Fig. 6A~~7~~, 102 may be deleted first in the 1st execution , and 67 may be deleted second in the 1st execution .

[0082] In the 2nd execution, for depth 3 of a left child node, the objects deleted and generated in the previous execution (1st) (i.e., at depth 4) may again be deleted and regenerated. Then, the objects 72 bytes and 32 bytes, having a depth 3, may be deleted and regenerated.

[0083] The 3rd execution, for depth 2 of the left child node, may be executed similar to the 2nd execution. In the 3rd execution, the objects deleted and generated in the previous execution (2nd execution) may again be deleted and regenerated, and then the object with 64 bytes and a depth 2 may be deleted and regenerated. The 3rd execution may complete the ~~deletion and regeneration of~~ the original left branch of the original binary tree data structure.

[0084] The 4th execution may be executed similar to the previous executions. . The objects deleted and regenerated in the previous execution (3rd execution) may again be deleted and regenerated, and then the object with 128 bytes (i.e., the boundary object) of the root

node may be deleted and regenerated. The 4th execution may complete the deletion and regeneration of the original root node of the original binary tree data structure.

[0085] In turn, objects of the right child node to the root node may then be deleted and regenerated. The 5th execution may be executed similar to the previous executions. In the 5th execution, for depth 2 of the right child node, the objects, deleted and generated in the previous execution (4th execution) may again be deleted and regenerated, and then the objects with 512 and 234 bytes, respectively, and with depth 2, may be deleted and regenerated.

[0086] The 6th execution may be executed similar to the previous executions. In the 6th execution, for depth 3 of the right child node, the objects deleted and generated in the previous execution (5n execution) may again be deleted and regenerated, and then the objects with 256 bytes at depth 3 may be deleted and regenerated. The 6th execution may complete the deletion and regeneration of the original right branch of the original binary tree data structure.

[0087] Referring to FIG. 6B, which may correspond to the binary tree data structure illustrated in FIG. 5B, illustrates the process of deletion and regeneration. However, as previous indicated, the deletion and regeneration may first be executed on the right sibling node instead of the left sibling node. FIG. 6B, in contrast to FIG. 6A, illustrates an order of deletion and regeneration first executed on the right sibling node instead of the left sibling node.

[0088] Referring to FIG. 6B, in a 1st execution, the object of the left child node with the deepest depth 5, 32 bytes, may be deleted and regenerated.

[0089] In a 2nd execution, the object of 64 byte in the left child node with depth 4 may be deleted and regenerated after object 32 bytes of the previous execution is again deleted and regenerated.

[0090] The 3rd execution may be executed similar to the previous execution. It should be noted that the object 102 may be deleted and

regenerated before object 67. If the deletion and regeneration process were executed similar to FIG. 6A, the object 67 would be deleted before the object 102, as object 102 is the right sibling and object 67 is the left sibling.

[0091] The 4th execution may be executed similar to the previous executions. The 4th execution may complete the deletion and regeneration of the original left branch of the original binary tree data structure.

[0092] The 5th execution may be executed similar to the previous executions. The 5th execution may complete the deletion and regeneration of the original root node of the original binary tree data structure.

[0093] The 6th execution may be executed similar to the previous executions.

[0094] The 7th execution may be executed similar to the previous executions. The 7th execution may complete the deletion and regeneration of the original left branch of the original binary tree data structure.

[0095] The 8th execution may be executed similar to the previous executions. The 8th execution may complete the deletion and regeneration of the original right branch of the original binary tree data structure.

[0096] As illustrated by FIGs. 6A and 6B, the smaller objects (of the left child nodes) may be more frequently deleted and regenerated as compared to the larger objects (of the right child nodes).

[0097] In an exemplary embodiment of the present invention, the JVM may include a multi-thread application. The multi-thread application may enable the binary tree data structure to have a plurality of instantiations with a plurality of formations. In this exemplary embodiment of the present invention, the multi-thread application may form a survival object. Survival objects may not be deleted when the program is running. Boundary objects in a thread of the multi-thread

application may be identical or different boundary objects of another thread of the multi-thread application.

[0098] FIG. 7 illustrates a process of generating benchmark files according to another exemplary embodiment of the present invention.

[0099] The process may be initiated in S701, and advances to S703. In S703, it may be determined whether there is a need for generating a survival object. The survival object may continuously occupy a memory without being deleted during an execution of a program. If there is determined to be a need for the survival object in S703, the survival object may be generated in S705. The survival object may have various data structures which may include link lists, arrays, a binary tree data structure, and/or other data structures. The JVM may use the survival object multiple times during the execution of a multi-thread application.

[00100] In S707, the process may determine the number of threads, the boundary object, the largest object, and/or the ratio between the generated objects larger than the boundary object and generated objects smaller than the boundary object. The boundary object may be determined to meet some criteria. Objects other than the boundary objects may be determined randomly.

[00101] A plurality of threads may be formed in S709. Each thread may comprise a binary tree data structure which may comprise objects.

[00102] In S711, the deletion and regeneration for objects may be executed in each thread. This process of deletion and regeneration may be similar to the process of deletion and regeneration described with respect to FIG. 6A or 6B.

[00103] At least one result of the execution of the threads may be reported in S713.

[00104] According to the exemplary embodiments of the present invention as described above, the present invention may provide a garbage collection benchmark.

[00105] The exemplary embodiments of the present invention being thus described, it will be obvious that the same may be varied in many ways. For example, the example source code and function names may be changed significantly while remaining within the scope of the invention. Further, the process of branching has been described first branching left and then branching right, however this process could easily be reversed. For example, objects less than a boundary object could branch to the right and objects greater than a boundary object could branch to the left.

[00106] For example, the functional blocks in FIGS. 1-7 describing the exemplary apparatus and methods may be implemented in hardware and/or software. The hardware/software implementations may include a combination of processor(s) and article(s) of manufacture. The article(s) of manufacture may further include storage media and executable computer program(s). The executable computer program(s) may include the instructions to perform the described operations or functions. The computer executable program(s) may also be provided as part of externally supplied propagated signal(s).

[00107] Further, such programs, when recorded on computer-readable storage media, may be readily stored and distributed. The storage medium, as it is read by a computer, may thus enable the exemplary embodiments of the benchmarking system and method described above.

Such variations are not to be regarded as departure from the spirit and scope of the exemplary embodiments of the present invention, and all such modifications as would be obvious to one skilled in the art are intended to be included within the scope of the following claims.

What is claimed is:

1. A method of benchmarking, comprising:

generating a first plurality of objects within a benchmark file;

deleting the first plurality of objects and regenerating a second plurality of objects in place of the deleted first plurality of objects;

executing the benchmark file; and

reporting at least one result of the execution.

2. The method of claim 1, wherein the first plurality of objects are generated in a binary tree data structure based on a size of at least one object of the first plurality of objects and the second plurality of objects are regenerated in the binary tree data structure based on the first plurality of objects.

3. The method of claim 1, wherein executing the benchmark file is performed in a Java virtual machine.

4. The method of claim 2, further comprising generating additional binary tree data structures with a same number of nodes but different binary tree data structures than the binary tree data structure with the first plurality of objects.

5. The method of claim 2, wherein generating the first plurality of objects further comprises:

branching two potential descendant nodes from a first parent node,

wherein an object with a size being less than or equal to an object of the first parent node is branched to a left potential descendant node, and an object with a size being greater than the object of the first parent node is branched to a right potential descendant node, and

wherein a second parent node not including a third parent node is a root node.

6. The method of claim 5, wherein a size of a regenerated second object is the same as a corresponding deleted first object, and wherein deleting the first plurality of objects and regenerating the second plurality of objects is carried out in the order of a descendant node branching left from the root node, the root node and a descendant node branching right from the root node.

7. The method of claim 6, wherein deleting the first plurality of objects and regenerating the second plurality of objects in a current node is carried out in an order beginning with a deepest node and ending with the current node.

8. The method of claim 6, further comprising generating additional binary tree data structures with a same number of nodes but different binary tree data structures than the binary tree data structure with the first plurality of objects.

9. The method of claim 2, wherein generating the first plurality of objects in the binary tree data structure comprises:

generating a root node for the binary tree data structure;

generating random numbers for the first plurality of objects;

branching two potential descendant nodes from the root node by using the random numbers to generate the binary tree data structure;

~~wherein~~ a random number being less than or equal to a size of the root node is branched to a left potential descendant node, and a random number being greater than the size of the root node is branched to a right potential descendant node, and a child node being less than or equal to a parent node is branched to a left of the patent node, and a

child node being greater than the parent node is branched to a right of the parent node.

10. The method of claim 2, further comprising generating survival objects prior to generating the benchmark file.

11. The method of claim 1, wherein the at least one result indicates at least one of sizes of allocated memory, sizes of unallocated memory, and time elapsed during the benchmark file execution.

12. The method of claim 4, further comprising analyzing object characteristics in real object oriented programs before generating the benchmark file, wherein the object characteristics comprise at least one of the number of objects, the average object size, the size of the objects frequently generated, the object with the largest size, and the object with the smallest size.

13. The method of claim 12, wherein a size of a root node is the average object size, and the size and number of the first plurality objects are determined based on the result of analyzing the object characteristics.

14. A method of benchmarking, comprising:

generating a first plurality of objects in a binary tree data structure within a benchmark file;

deleting the first plurality of objects and regenerating a second plurality of objects in place of the deleted first plurality of objects;

executing the benchmark file; and

reporting at least one result of the execution.

15. The method of claim 14, wherein generating the first plurality of objects further comprises:

branching two potential descendant nodes from a first parent node,

wherein an object with a size being less than or equal to an object of the first parent node is branched to a left potential descendant node, and an object with a size being greater than the object of the first parent node is branched to a right potential descendant node, and

wherein a second parent node not including parent node is a root node.

16. The method of claim 15, wherein a size of a regenerated second object is the same as corresponding deleted first object, and wherein deleting the first plurality of objects and regenerating the second plurality of objects is carried out in an order of a descendant node branching left from the root node, the root node and a descendant node branching right from the root node.

17. The method of claim 16, wherein deleting the first plurality of objects and regenerating the second plurality of objects in a current node is carried out in an order beginning with a deepest node and ending with the current node.

18. The method of claim 16, further comprising generating additional binary tree data structures with a same number of nodes but different binary tree data structures than the binary tree data structure with the first plurality of objects.

19. The method of claim 14, further comprising analyzing object characteristics in real object oriented programs before generating the benchmark file, wherein the object characteristics comprise at least one of the number of objects, the average object size, the size of the objects frequently generated, the object with the largest size, and the object with the smallest size.

20. The method of claim 19, wherein a size of a root node is the average object size, and the size and number of the first plurality objects are determined based on the result of analyzing the object characteristics.

21. A system of benchmarking, comprising:

a processor for generating a first plurality of objects of a binary tree data structure in a benchmark file, for deleting the first plurality of object and regenerating a second plurality of objects in the binary tree data structure, for executing the benchmark file, and for reporting a result of the execution.

22. The system of claim 21, wherein said processor further comprises:

branching two potential descendant nodes from a first parent node,

wherein an object with a size being less than or equal to an object of the first parent node is branched to a left potential descendant node, and an object with a size being greater than the object of the first parent node is branched to a right potential descendant node, and

wherein a second parent node not including a third parent node is a root node.

23. The system of claim 22, wherein each of the regenerated second plurality of objects are the same size as each of the first plurality of objects, and wherein the deleting and regenerating is carried out in an order of a descendant node branching left from the root node, the root node and a descendant node branching right from the root node.

24. The system of claim 23, wherein deleting the first plurality of objects and regenerating the second plurality of objects in a current

node is carried out in an order beginning with a deepest node and ending with the current node.

25. The system of claim 21, wherein the processor generates additional binary tree data structures with a same number of nodes but different binary tree data structures than the binary tree data structure by the first plurality of objects.

26. The system of claim 21, wherein the processor generates survival objects prior to generating the benchmark file.

27. A computer program comprising a computer-readable medium having computer program logic stored thereon for enabling a data processing unit to perform benchmarking, comprising:

generating a first plurality of objects in a binary tree data structure within a benchmark file;

deleting the first plurality of objects and regenerating a second plurality of objects in place of the deleted first plurality of objects;

executing the benchmark file; and

reporting at least one result of the execution.

28. The computer program of claim 27, wherein the at least one result indicates garbage collection performance of the benchmark file.

29. The computer program of claim 27, wherein executing the benchmark file is performed in a Java virtual machine.

30. The computer program of claim 27, wherein each first parent node in the binary tree data structure includes two potential descendant nodes and a second parent node not including a third parent node is a root node,

wherein one of the two potential descendant nodes includes a first object with a size less than or equal to a second object of the first parent node is branched left from the first parent node, and the other of the two potential descendant nodes including a third object with a size greater than the object of the parent node is branched right from the parent node,

wherein deleting the first plurality of objects and regenerating the second plurality of objects is carried out in an order of a descendant node branching left from the root node, the root node and a descendant node branching right from the root node.

31. The method of claim 30, wherein deleting the first plurality of objects and regenerating the second plurality of objects in a current node is carried out in an order beginning with a deepest node and ending with the current node.

32. A system of benchmarking for performing the method of claim 1.

33. A computer program for performing the method of claim 1.

34. A system of benchmarking for performing the method of claim 14.

35. A computer program for performing the method of claim 14.

Application No: GB0418969.2 **Examiner:** Robin Stout

Claims searched: 1 to 35 **Date of search:** 23 December 2004

Patents Act 1977: Search Report under Section 17

Documents considered to be relevant:

Category	Relevant to claims	Identity of document and passage or figure of particular relevance
X	1, 14, 21, 27 at least	Wikipedia, "Binary Search Tree", earliest version 14 June 2002, found at http://en.wikipedia.org/wiki/Binary_search_tree
X	1, 14, 21, 27 at least	WO 01/73556 A1 (TAO) See abstract, page 10 line 2 to page 11 line 16, figure 3.
X	1, 14, 21, 27 at least	WO 00/41079 A3 (PHILIPS) See abstract, whole document.
X	1, 3, 14, 21, 27, 29	US 6118940 A (IBM) See column 4 line 53 to column 5 line 5.
X	1, 14, 21, 27 at least	US 6070173 A (IBM) See column 3 lines 8 to 55.

Categories:

X	Document indicating lack of novelty or inventive step	A	Document indicating technological background and/or state of the art.
Y	Document indicating lack of inventive step if combined with one or more other documents of same category.	P	Document published on or after the declared priority date but before the filing date of this invention.
&	Member of the same patent family	E	Patent document published on or after, but with priority date earlier than, the filing date of this application.

Field of Search:

Search of GB, EP, WO & US patent documents classified in the following areas of the UKC[W] :

G4A

Worldwide search of patent documents classified in the following areas of the IPC[07]

G06F

The following online and other databases have been used in the preparation of this search report

EPODOC, WPI, JAPIO, INSPEC, Internet